UCRL-PRES-209445

# Parallelizing the Point-Clustering Algorithm in Structured Adaptive Mesh Refinement

**Brian Gunney and Andy Wissink**

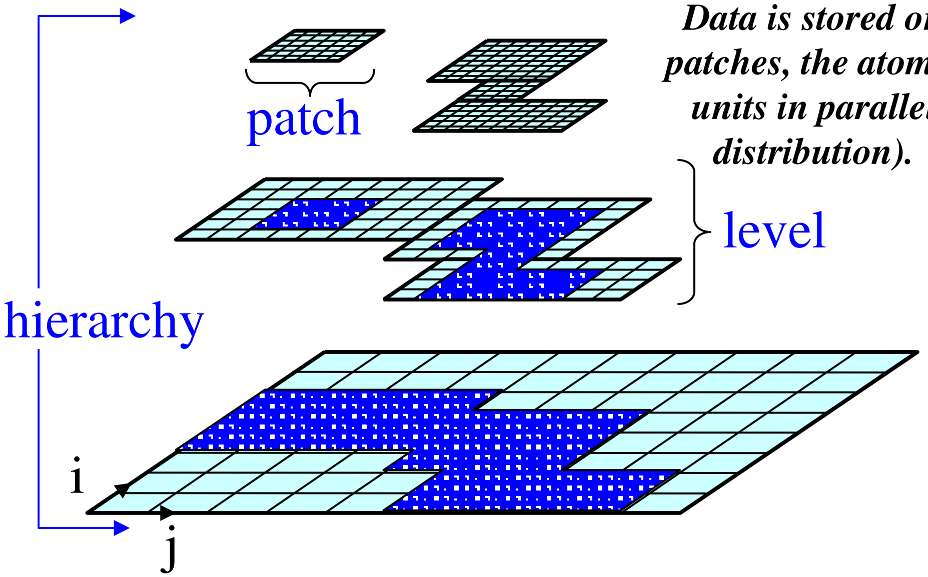
**Center for Applied Scientific Computing**
**Lawrence Livermore National Laboratory**

**SIAM CSE Conference**
**15 February 2005**

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

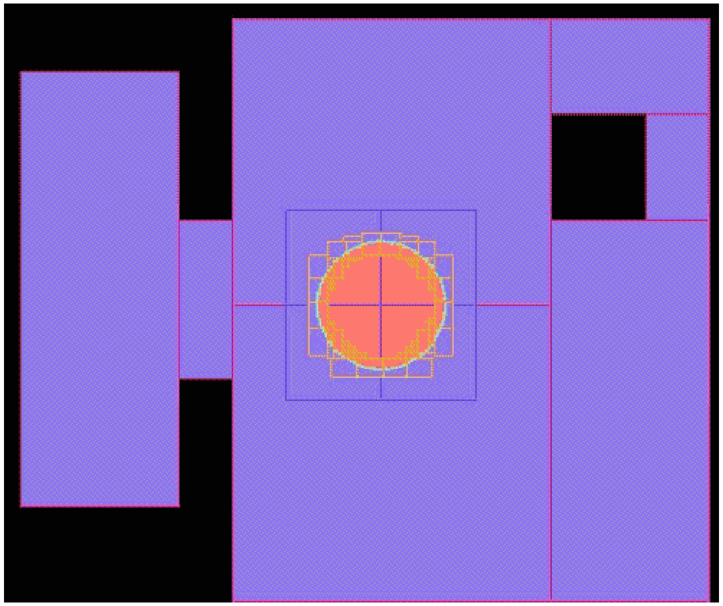
# Global nature of structured AMR mesh adaption presents scalability challenges

## Hierarchical mesh



*Data is stored on patches, the atomic units in parallel distribution).*

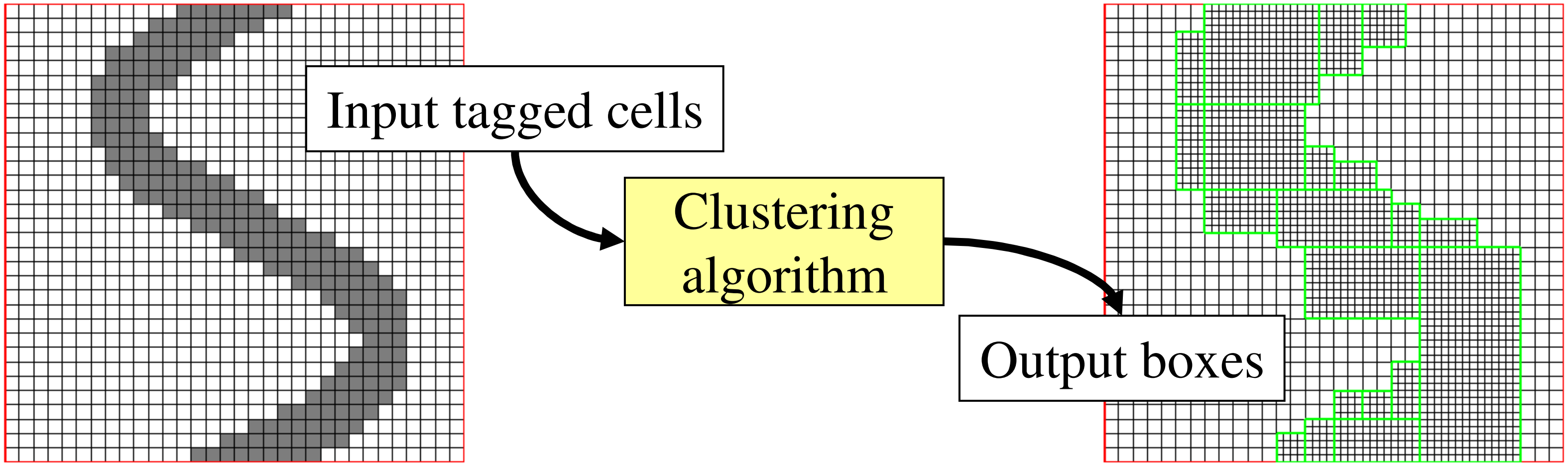
## Simulation



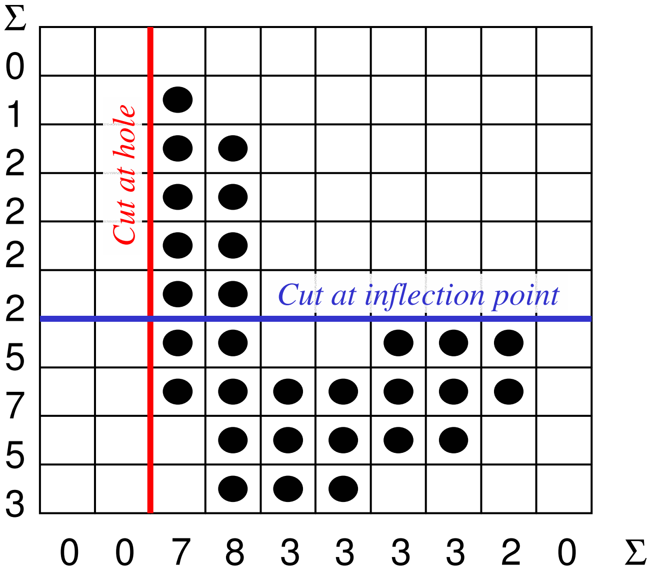
## Scalability challenges

1. Dynamic meshes are adapted throughout simulation.
2. Computation generally uses nearest neighbor data. Grid adaption involves the entire grid.
3. LLNL’s BG/L machine will have 65K processors

# Grid adaption uses a recursive bisection algorithm to cluster tagged cells



1. **Start with bounding box covering whole domain**
2. **Form signature (Σ) for box**
3. **If there are too many untagged cells**
   - a. **Cut at hole or inflection point to form sub-boxes**
   - b. **Repeat on each sub-box**



**Berger and Rigtousos, IEEE Transactions on Systems, Man and Cybernetics, Vol. 21, No. 5, 1991.**

# Recent implementations of clustering algorithm in SAMRAI use SPMD model

## Global communication:

```
fcn(box) {
  h = signature(box)
  if (h is acceptable)
    accept(box)
  else
    split(h,left,right)
    fcn(left)
    fcn(right)
}
```

1. signature() includes global sum-reduce.
2. accept() and split() are local operations.
3. All processors run algorithm.
4. All processors have outputs.

## Grouped communication:

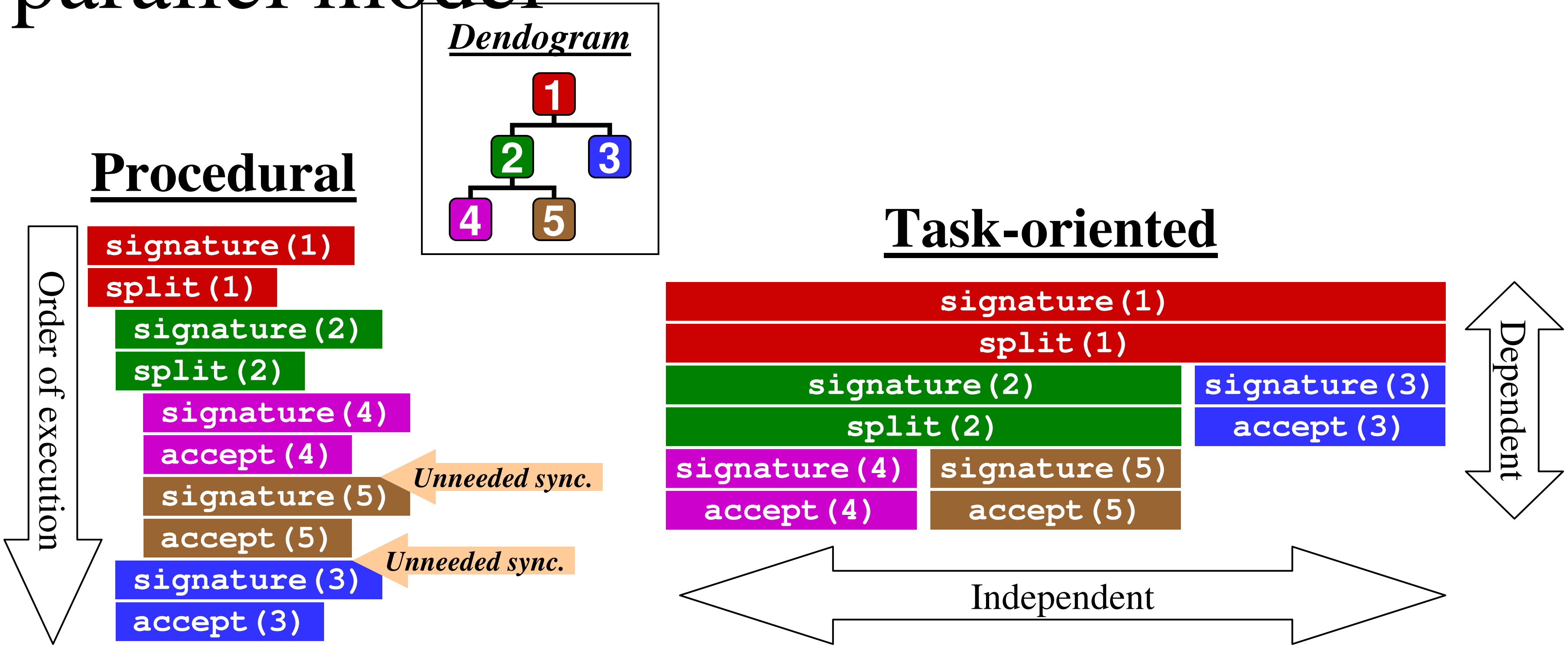
```
fcn(box) {
  h = signature(box)
  if (h is acceptable)
    accept(box)
  else
    split(h,left,right)
    if (overlap(left )) fcn(left)
    if (overlap(right)) fcn(right)
}
```

*Functions in red require communication (synchronization)*

1. signature() includes sum-reduce to processor 0.
2. accept() and split() include broadcast from processor 0.
3. Only processor 0 runs full algorithm.
4. *Only processor 0 has outputs, which must be globalized using a broadcast.*

Wissink, Hysom and Hornung, Proceedings of 17th ACM Int'l Conf. on Supercomputing, 2003.

# Clustering algorithm contains independent paths appropriate for task-parallel model



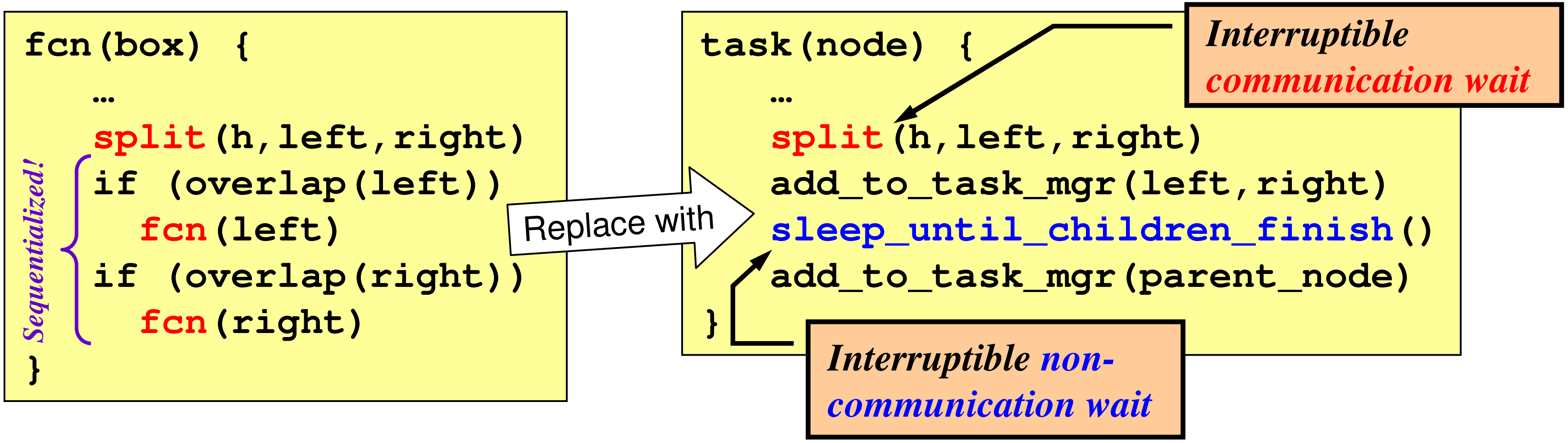
**Procedural**

1. Impose specific ordering.
2. Create unneeded synchronizations.

**Task-oriented**

1. Each dendogram node is a task.
2. Do independent tasks concurrently.
3. Overlap communications and computations in multiple tasks.

# Task-parallel implementation follows natural tasks in clustering algorithm



**Sequential, old**

```
fcn(box) {
    …
    split(h,left,right)
    if (overlap(left))
      fcn(left)
    if (overlap(right))
      fcn(right)
}
```

*Sequentialized!*

Replace with

**Task-oriented, new**

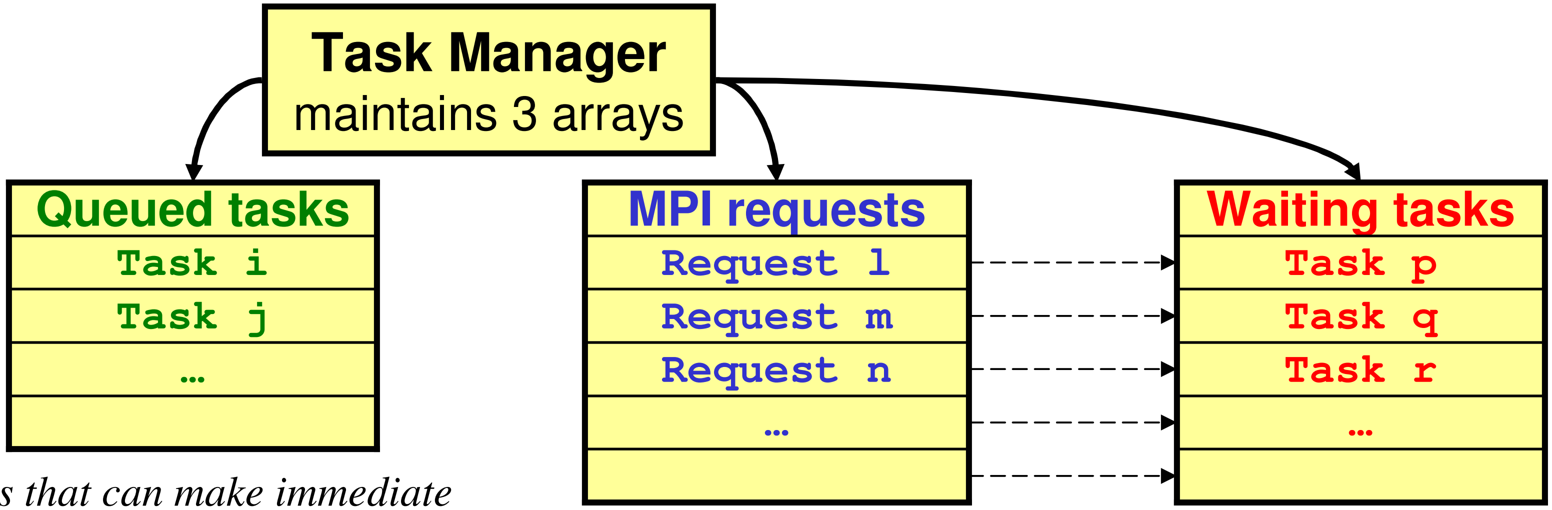
```
task(node) {
    …
    split(h,left,right)
    add_to_task_mgr(left,right)
    sleep_until_children_finish()
    add_to_task_mgr(parent_node)
}
```

***Interruptible communication wait***

***Interruptible non-communication wait***

- Replace sequential operations on left and right children with insertion into task manager (next slide).
- Each instance of `task()` is associated with one node of the dendogram. It is *not* recursive.
- Communication and sleeping steps are “interruptible” so waiting tasks can be set aside to work on tasks that can make immediate progress.
- Tasks are initiated by insertion into task manager (not directly called).
- Task-parallel algorithm driven by task manager (next slide).

# Task manager selects active tasks to minimize processor wait times



**Task Manager**
maintains 3 arrays

| Queued tasks |
|---|
| `Task i` |
| `Task j` |
| … |
| |

*Tasks that can make immediate progress without waiting (new and awoken task)..*

| MPI requests |
|---|
| `Request l` |
| `Request m` |
| `Request n` |
| … |
| |

*Pending communications (MPI_Request objects from non-blocking MPI calls).*

| Waiting tasks |
|---|
| `Task p` |
| `Task q` |
| `Task r` |
| … |
| |

*Tasks awaiting communication from* `signature()`, `split()`, `accept()`, …

**Task Manager Algorithm (user-space thread controller):**

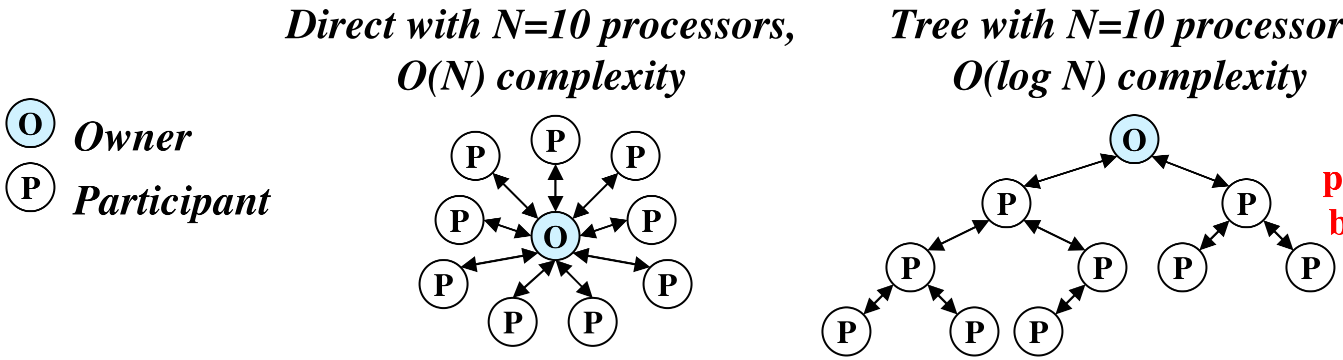
1. Start/continue all tasks in **task queue** (and empty the queue)
2. Wait for some **pending communication requests** to complete
3. Continue **waiting tasks** for completed communications
4. Repeat until no more task or pending request exists.

# Two optimizations affect new algorithm

**Multi-owner option – select owner from participating processor group**

- **Reduces traffic congestion around single-owner**
- **Improves load balance**
- ***Requires all-gather instead of broadcast for globalizing output (drawback).***

**Hand-coded collective communications along edges of a tree**



*Direct with N=10 processors, O(N) complexity*

*Tree with N=10 processors, O(log N) complexity*

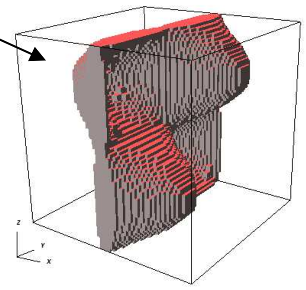
**(Tree of processors, not to be confused with dendogram!)**

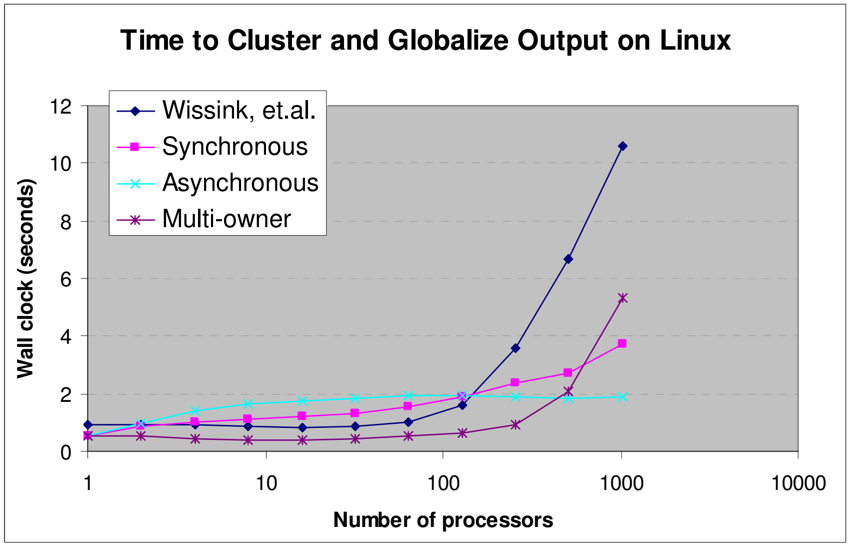
- **Avoid expensive formation of MPI communicators.**
- **Supports non-blocking collective communications.**

# New algorithm improves clustering performance and scalability

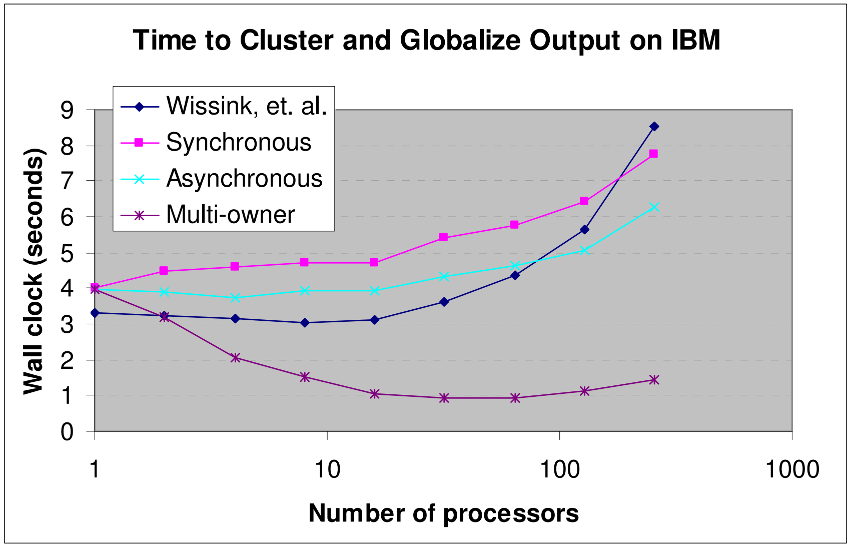
- Adapt four-level mesh around moving sinusoidal wave front.
- Fix problem-size. Increase processor count.
- Synchronize processors before clustering.
- Globalize (put all outputs on all processors).
- Time initial mesh generation and five global regrids.
- Collect max times across all processors.

***Tagged cells***





**Linux cluster: 2 processors/node**



**IBM: 16 processors/node**

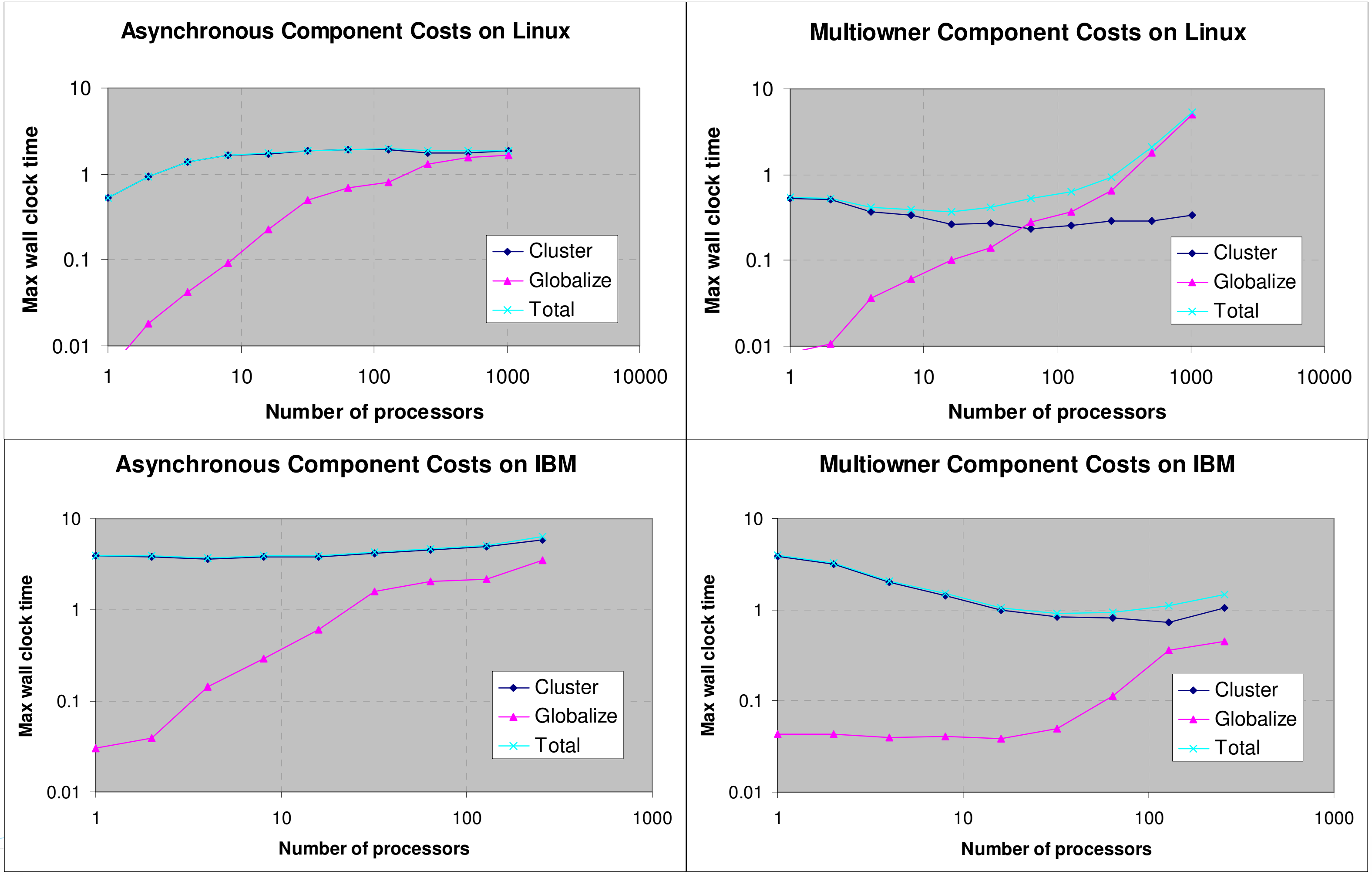
# Clustering scales better than output globalizing

Asynchronous

Multiowner

Linux cluster

IBM



# Summary

- This work aims to reduce the regridding cost of SAMR on large-scale parallel computers.
- We parallelized the clustering algorithm by wrapping the SPMD implementation an asynchronous, interruptible tasks. A task manager selects active tasks to minimize communication wait times.
- Task-parallel implementation significantly improves scaling trend over the synchronous implementations
  - Scaling trend is much more favorable
  - Performance with low processor-count is comparable to synchronous algorithm, but is machine dependent.
  - Different variations of the implementation works better with different platforms (and underlying MPI implementations).
- Clustering cost scales much better than output globalizing cost.